A Declarative Approach to Multi-Layer Path Finding Based on Semantic Network Descriptions

Li Xu*, Freek Dijkstra†, Damien Marchal*, Arie Taal*, Paola Grosso*, Cees de Laat*
* System and Network Engineering Group
Universiteit van Amsterdam
Science Park 107, 1098 XG Amsterdam, The Netherlands
Email: {l.xu, d.marchal, a.taal, p.grosso, delaat}@uva.nl
† SARA Computing & Networking Services
Science Park 121, 1098 XG Amsterdam, The Netherlands
Email: freek.dijkstra@sara.nl

***Abstract*—With the increasing demand for dynamic network connections between multiple research networks, a number of issues on multi-layer hybrid networks need to be addressed, such as network representation, path finding, and path provisioning. This paper focuses on solving the multi-layer path finding problem by a declarative approach. The declarative approach uses Prolog and is compared to an imperative algorithm in Python that was published earlier. We present semantic web technologies to describe the multi-layer network and introduce the complexity of the path finding problem with a typical multi-layer network example.**

I. Introduction

As worldwide distributed research collaboration the need for deterministic network connectivity amongst institutions is rising. The implementations typically make use of so-called light paths [1]. Given that technologies evolve over time and network administrators each make their own decision about the technology to use, these interconnected networks will be multi-technology in nature. Multi-technology networks can be modelled as multi-layer networks where the configuration of the network can be changed at multiple layers. Examples of these networks include optical and hybrid networks where the TDM (SDH/SONET), WDM, OTN, and Ethernet layers can be dynamically reconfigured. The integration of the different technologies in transport networks increases the complexity of path finding.

The path finding problem in single layer networks is well studied, while path finding in multi-layer networks, such as multi-technology networks, is far from trivial. Technology incompatibilities in networks lead to complex constraints in path finding algorithms. The algorithms that are used in single layer routing protocols, such as the Ford-Fulkerson algorithm [2] for BGP and Dijkstra's algorithm [3] for OSPF, can not deal with the complexity of the multi-layer networks and fail to find the shortest multi-layer path. In order to solve this problem, both a multi-layer network model description and new path finding algorithms need to be developed.

Some pilot researches have been carried out using the imperative programming language Python [4], [5] . The major limitations of these approaches are the high complexity of implementation of the algorithm and the incompatibility of the non-network resources (e.g. computing nodes). In this paper, we compare two alternative solutions to the multi-layer path finding problem, an imperative program in Python (as a benchmark) and a declarative program in Prolog.

This paper is organized as follows. In the next section we introduce the technology used to describe multi-layer networks. We will then elaborate on the complexity in multi-layer network path finding. In section IV, we will present our implementations by using different approaches and show some comparisons and discussions in section V. Finally we propose the future work and draw our conclusions.

II. Multi-Layer Network Description

The Network Description Language (NDL) is a RDF ontology of optical hybrid networks [6]. It helps network and service providers to efficiently describe their network resources, and exchange this information among each other. This information can be used for visualization and path finding [7]. NDL is used in the community of national research and education networks for these applications. NDL provides several schemata that can be used for this purpose: a topology schema that describes the basic interconnections between devices, a layer schema to describe technologies, and a capability schema to describe network capabilities and a domain schema for creating abstracted views of networks.

NDL is strongly based on the functional elements defined by ITU-T recommendations G.805 and G.800 [8], [9]. In addition, the G.805 and G.800 model is extended so NDL can not only be used to describe the state of a network, but also its capabilities – how the state can be changed, and by whom [10].

A unique selling point of NDL is that its base is completely technology independent. This property has been used to create a path finding implementation that does not need to be changed as new technologies come along [4].

III. Complexity of Multi-Layer Path finding

A. Multi-Layer Path Finding

Multi-layer path finding is fundamentally different from single layer path finding. A shortest path in a multi-layer network may be looped (traverse the same link twice), and

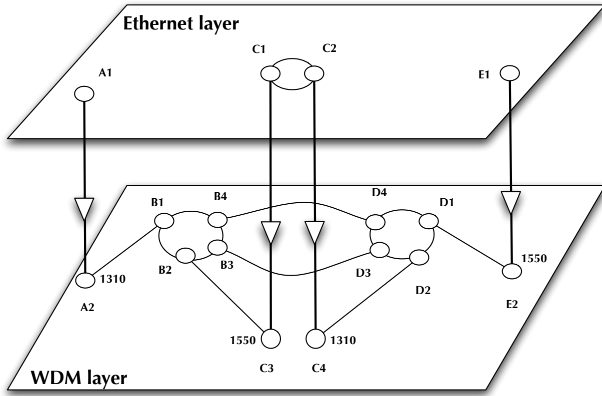

Fig. 1. The example of multi-layer network.

a segment of a shortest path may not be a shortest path in itself [4], [5].

Fig.1 shows a graphical representation of a sample multi-layer network. This representation uses functional elements and logic as defined in ITU-T recommendations G.805 and G.800 [8], [9]. This network consists of five devices: *A*, *B*, *C*, *D* and *E*. Each device is modelled as a switch matrix with logical interfaces (*Forwarding Points* in G.800 terminology), the circles in Fig.1. Devices *B* and *D* are wavelength selective switches that can forward a wavelength of a specific color to another wavelength. As these devices have pure WDM technology interfaces, they are represented in the WDM-layer only. Devices *A*, *C*, and *E* are Ethernet devices, with interfaces operating at a fixed wavelength (either 1310 or 1550 nm). For example, device *C* has two interfaces and can forward Ethernet traffic between logical interfaces *C1* and *C2*. The Ethernet packets of these two interfaces are adapted into a specific wavelength before those are forwarded to neighboring devices. This is indicated by interface nodes in both layers with an adaptation (edge with triangle symbol) between them. The Ethernet data of logical interface *C1* is adapted into *C3* which is a laser operating at 1550 nm, while the Ethernet data of *C2* is adapted into *C4*, a laser operating at 1310 nm.

The wavelength selective switches B and D can not convert between wavelengths; they forward the signal as-is. Two forwarding end points at the WDM layer can only exchange data if they operate at the same wavelength. A network connection between *A2* and *E2* would result in a failure, while a network connection between *A2* and *C4* would be feasible. Following the logic in the G.805 and G.800 recommendations, a network connection on the WDM layer, with adaptation at the source and de-adaptation at the sink, would result in a link connection at the Ethernet layer.

Our goal is to find the shortest feasible path from *A1* to *E1*. Since *A2* and *E2* have incompatible wavelengths, the wavelength has to be converted at device *C*. The shortest path is thus $A1 - A2 - B1 - B3 - D3 - D2 - C4 - C2 - C1 - C3 - B2 - B4 - D4 - D1 - E2 - E1$. An alternative shortest path first traverses $B4 - D4$ before $B3 - D3$. This solution could also be found by a standard shortest-path algorithm if edges and interface nodes for each wavelength are duplicated, but this is not possible in the general case and does not scale either [4].

B. Link constrained vs. Path constrained

The edges $B3 - D3$ and $B4 - D4$ represent link connections at the WDM layer, thus individual wavelengths. It is not specified how these wavelengths are transported between devices *B* and *D*. It could be two physical links to one physical link with two wavelengths, using wavelength division multiplexing. If it is the later option, the shortest path would traverse the same physical link twice.

Multi-layer network connections are fundamentally different from single layer network connections. Whereas constraints in single layer networks apply to individual edges or nodes, the constraints in multi-layer networks may depend on the combination of multiple edges or nodes [4], [5]. For example, the use of the interface *A2* or the interface *E2* are perfectly valid, but the combination of the two in a network connection is not.

IV. Implementations

A. Path Finding Approaches

In earlier work we have shown a technology-independent multi-layer network model and the properties of a valid path through such a network [10]. How to obtain a valid path given a network, source and destination is another matter.

We have proposed a path finding algorithm based on a breadth first search algorithm [5]. This is a hop-by-hop algorithm: it starts with the source, and finds neighboring nodes to which it can extend the path. It then repeats this process starting with these neighboring nodes. Neighbors can be connected using links, adaptations, de-adaptations or internal cross connects inside devices. We will present an implementation of this algorithm in the next section.

An alternative approach to the path finding is a layer-by-layer algorithm. Rather than examining direct neighbors, it examines all other nodes on the same layer, and checks which of these nodes can reach each other. There are two variants of the layer-by-layer approach: a bottom-up approach that starts at the lowest layer and a top-down approach that starts at the highest layer.

The bottom-up layer-by-layer approach uses a forward chaining reasoning engine. It takes all possible network connections on the lowest layer and deduce link connections at a higher layer from these. In our example network in Fig.1, the links at the WDM layer lead to valid network connections between *A2* and *C4*, and between *C3* and *E2*. That corresponds to valid link connections between *A1* and *C2* and between *C1* and *E1* at the Ethernet layer. The disadvantage of this approach is that it is untargeted: it will find all possible connections, without looking for the specific request for a network connection between a given source and destination.

The top-down layer-by-layer approach uses a backward chaining reasoning engine. It starts with the source and

destination at the highest layer, and tries to find network connections at lower layer that would result in the given connections at a higher layer.

B. Imperative programming

Earlier we implemented and described a hop-by-hop path finding algorithm in Python [4].

The Python implementation makes use of Python NDL Tools (informally *Pynt*), a software library designed to read, write and process NDL files. At the core is a large set of Python objects representing computer networks and network technologies. The whole Pynt module is rather complex. For example, the switch matrix class can determine if a cross connect is possible or not, based on generic properties such as the ability to convert between labels or not (e.g. wavelength conversion), unicast, multicast or broadcast support of devices, and support for both unidirectional and bidirectional connections.

Fig.2 lists the number of classes and lines of code for key modules in Pynt. This does not include the import and export modules.

Module	Description	# Classes	# Lines
pynt.xmlns	Generic RDF classes	4 classes	507 lines
pynt.elements	Network description	19 classes	1560 lines
pynt.layer	Technology description	5 classes	308 lines
pynt.paths	Path descriptions	18 classes	289 lines
pynt.algorithm	Path finding algorithms (main variant)	2 classes	381 lines

Fig. 2. Code size for key modules in Pynt.

C. Declarative Programming

Section II describes the NDL as a network representation in RDF. The use of RDF permits the harvesting and manipulation of network data, much like it is stored in a database. SPARQL is a SQL-like query language to explore information in an RDF database. For example, the query in Fig.3 searches in the database for two devices d1 and d2 that are connected by a path that consists of two links (an NDL `connectedTo` statement) and a cross connect (an NDL `switchedTo` statement).

If a query can be used to find a path of length 3, could it be extended to find a path of any length? To achieve this, it is necessary to employ a 'query language' that allows recursive statements and is more expressive than SPARQL. We choose to use Prolog. Prolog was introduced in the early 70's; and unlike many other programming languages it is based on a declarative syntax. In Prolog a program is expressed

```
PREFIX ndl: <http://www.science.uva.nl/research/sne/ndl#>
SELECT ?d1 ?d2
WHERE
{
        ?d1 ndl:hasInterface ?i1.
        ?i1 ndl:connectedTo ?i2.
        ?i2 ndl:switchedTo ?i3.
        ?i3 ndl:connectedTo ?i4.
        ?d2 ndl:hasInterface ?i4
}
```

Fig. 3. A SPARQL query can search for two devices *d1, d2* that are connected to the same switch.

in terms of logical relations (rules) between predicates[1] and the program execution is equivalent to search for the satisfaction of these logical relations. Presently, there are several Prolog implementations available. One of them, SWI-Prolog [11], provides libraries for RDF management and semantic manipulation. These libraries can load and manipulate an RDF database stored in the triplet format: *Subject - Predicate - Object*. Fig.4 defines rules to transform the NDL statements into roughly equivalent Prolog predicates. The `canswitchto()` rule has no equivalent NDL statement is introduced here for simplicity. Whereas an `switchedTo()` predicate asserts that a cross connect exists, a `canswitchto()` predicate asserts that it is possible to create a cross connect. For readability each rule is identified: R1, R2, R3, and so on.

```
R1: linkto(Intf1, Intf2):-
    rdf_db:rdf(Intf1, ndl:'linkTo', Intf2).
R2: adapts(Intf1, Intf2):-
    rdf_db:rdf(Intf1, ndllayer:'adapts', Intf2).
R3: isadaptedin(Intf1, Intf2):-
    rdf_db:rdf(Intf1, ndllayer:'isAdaptedIn', Intf2).
R4: switchmatrix(Device, Matrix):-
    rdf_db:rdf(Device, ndlcap:'hasSwitchMatrix', Matrix).
R5: canswitchto(X, Y):-
    hasinterface(S, X),
    switchmatrix(S),
    hasinterface(S, Y),
    X \= Y.
```

Fig. 4. Rules that map the NDL statements stored in the RDF database to Prolog predicates.

Fig.5 indicates how the NDL predicates are used to implement a simple path finding program. The following logical relationships are defined: X and Y are a neighbor if there is a link (PR1), a switchmatrix (PR2) or an adaptation (PR3). There is a path between X and Y if either X and Y are a neighbor (PR4), or iX and Z are a neighbor and there is a path between Z and Y (PR5). Note that the `[Z | Path]` construct in Prolog is equivalent to a list with element Z appended to list Path. This logic defines all possible paths that consist of

[1]A Prolog predicate is roughly equivalent with an RDF statement, and is unrelated to an RDF predicate.

```
PR1: neighbor(X, Y):-          linkto(X, Y).
PR2: neighbor(X, Y):-          canswitchto(X, Y).
PR3: neighbor(X, Y):-          adapts(X, Y); isadaptedin(Y, X)     .

PR4: path_(X, Y, Visited, [Y] ):-
        neighbor(X, Y),
        not( member(Y, Visited) ).

PR5: path_(X, Y, Visited, [Z | NPath] ):-
        neighbor(X, Z),
        not( member(Z, Visited) ),
        path_(Z, Y, [Z|Visited], NPath ).

PR6: path(X, Y, [X | Path]):-
        path_(X, Y, [X], Path).
```

Fig. 5. Simple path finding algorithm.

consecutive neighbors. PR4 is the base case, whereas PR5 is the inductive step of the recursive definition.

Executing the path finding program consists in fact in proving that `path(X, Y, Path)` is true. To do so, the Prolog virtual machine (the reasoning engine) proceeds by backward chaining reasoning; starting from the query to prove that this is a consequence of known facts stored in the database. Consider the steps if Prolog is given the network of Fig.1 and asked for a path between A1 and C1. The predicate `path(a1, c1, Path)` is true if `path_(a1, c1, [a1], Path)` is true. While, according to PR4, `path_(a1, c1, [a1], Path)` is true if `neighbor(a1, c1)` is true. As this is not the case a backtracking operation is carried out to try the PR5 alternative instead of PR4: `path_(a1, c1, [a1], Path)` is true if `neighbor(a1, Z)` is true, `neighbor(a1, Z)` is true if `linkto(a1, Z)` is true, `linkto(a1, Z)` is true if `Z=a2` and so on. Similar to the Python implementation, the way Prolog solves a path query is by doing a hop-by-hop path finding, although this simple variant does not take any path constraints into account.

This simple algorithm can be extended to support additional constraints that arise in the multi-layer path finding. The multi-layer path finding algorithm in Prolog, as shown in Fig.6, is capable of that logic. The result of this algorithm is not a list of nodes, but a list of cross connects that must be made in order to create the requested path.

The algorithm closely follows the logic from the ITU-T G.805 specification, as given at the end of section II:

- lc: link connection, either direct linkto, or implied via adaptation and a network connection on a lower layer.
- nc: network connection, sequence of one or more link connections and/or subnetwork connections
- snc: subnetwork connections: a cross connect, or canswitchto() statement.

The definition of network connections is restricted to be either a link connection, or a sequence of link connections, subnetwork connections and network connections. These logical constrains that are also present in the NDL model, can be

```
nc(Src, Dst, Crss):- nc_dfs(Src, Dst, Crss, _, _, [], [], []).

lc_dfs(Src, Dst, [], [], [], _, _, _):- linkto(Src, Dst).

lc_dfs(Src, Dst,
        DownCrss, DownCrsSrcs, DownCrsDsts,
        UpCrss, CrsSrcs, CrsDsts):-
        adapts(Src, Hop1),
        isadaptedin(Hop2, Dst),
        label(Hop1, Label),
        label(Hop2, Label),
        Hop1 \= Hop2,
        nc_dfs(Hop1, Hop2,
                DownCrss, DownCrsSrcs, DownCrsDsts,
                UpCrss, CrsSrcs, CrsDsts).

nc_dfs(Src, Dst,
        DownCrss, DownCrsSrcs, DownCrsDsts,
        UpCrss, CrsSrcs, CrsDsts):-
        lc_dfs(Src, Dst, DownCrss, DownCrsSrcs, DownCrsDsts,
                UpCrss, CrsSrcs, CrsDsts).

nc_dfs(Src, Dst, DownCrss, DownCrsSrcs, DownCrsDsts,
        UpCrss, UpCrsSrcs, UpCrsDsts):-
        lc_dfs(Src, Hop1,
                DownCrss1, DownCrsSrcs1, DownCrsDsts1,
                UpCrss, UpCrsSrcs, UpCrsDsts),
        Cross = canswitchto(Hop1, Hop2),
        Cross,
        append(UpCrss, DownCrss1, NewUpCrss),
        append(UpCrsSrcs, DownCrsSrcs1, NewUpCrsSrcs),
        append(UpCrsDsts, DownCrsDsts1, NewUpCrsDsts),
        snc_dfs(Cross, NewUpCrss, NewUpCrsSrcs,
                        NewUpCrsDsts),
        nc_dfs(Hop2, Dst,
                DownCrss2, DownCrssSrcs2, DownCrssDsts2,
                [Cross | NewUpCrss], [Hop1 | NewUpCrsSrcs],
                [Hop2 | NewUpCrsDsts]),
        append(DownCrss1, [Cross | DownCrss2], DownCrss),
        append(DownCrsSrcs1, [Hop1 | DownCrssSrcs2],
                DownCrsSrcs),
        append(DownCrsDsts1, [Hop2 | DownCrssDsts2],
                DownCrsDsts).

snc_dfs(_, [], [], []).

snc_dfs(Cross, Crss, _, _):-   member(Cross, Crss).

snc_dfs(Cross, Crss, Srcs, Dsts):-
        Crss \= [],
        not( member(Cross, Crss)),
        Cross = canswitchto(Hop1, Hop2),
        not( member(Hop1, Srcs)), not( member(Hop1, Dsts)),
        not( member(Hop2, Dsts)), not( member(Hop2, Srcs)).
```

Fig. 6. The complete multi-layer path finding implementation in Prolog. lc = link connection; nc = network connection; dfs = depth first search.

implemented naturally in Prolog. The only constraint that is not trivial to implement is that a path cannot go twice through the same interface. This is handled by keeping a list of cross connects, and making sure that the list of valid cross connects does not cause conflicts (e.g. an interface can not be used in two different cross connects). Much of the complexity of the definitions in Fig.6 are caused by the requirement to both track and backtrack the list of cross connects: the tracking (called upstream) to make sure the depth first search has no infinite loops, and backtracking (called downstream) to present the list of used cross connects as the result of the algorithm.

During the execution of the Prolog query, the reasoning proceeds layer-by-layer. The Prolog rules are defined in such an order that a path is first examined at the given layer, and only if no path is available at this layer, a path is examined at the underlying layer. During this process, the consistency of the wavelength at transmit and receive is ensured. For path finding within the same layer, it uses a depth first search algorithm, starting at the source. The result of the presented algorithm is a list of the cross connects that must be made in order to create the network connection. The same algorithm is also capable to deal with the following queries:

```
Q1: nc('A1', 'E1', Path).
Q2: nc('A1', X, Path).
Q3: nc(X, Y, Path).
```

Q1 corresponds to the query: “Is there a path between A1 and E1?”, Q2 corresponds to the query: “What are the paths starting at A1?”, whereas Q3 corresponds to “What are all available network connections?”

V. Discussion

Finding a path in a multi-layer network is an NP-Complete problem [5], and all solutions we presented are exact.

The Prolog and Python implementations are equivalent as they are both Turing Complete, so we need other metrics to compare the difference. In [12], [13], the lines of code is a metric for implementation easiness. If we compare our two implementations, it is clear that an equivalent implementation using an imperative language like Python would require additional definitions of data-structures. More important is to note that the complete multi-layer path finding query in Prolog needs only 8 predicates and 56 lines of code whereas the multi-layer path finding in Python is roughly 3045 lines of code, including 48 classes. The comparison is a bit skewed, since the Python code has much more functionality, but even with that removed, the difference remains significant.

An interesting example of the advantage of declarative programming is that in Prolog all the parameters can be free variables and thus, with the same code, it is possible to issue multiple queries as we see in Fig.7. Doing the same thing in Python would require a large amount of additional code to handle all the different variants. For example, there is a query in which we request multiple paths that start on the same device forming a star-shape structure. Such kind of query is demanded to multi-cast a video stream for instance. The simplest way to fulfill it in Prolog is to add an additional constraint to ensure that the found paths are not sharing the same resources.

```
PQ1: path('A1', 'E1', Path).
PQ2: path('A1', Dst, Path).
PQ3: path(Src, 'E1', Path).
PQ4: path(Src, Dst, Path).
```

Fig. 7. Four differents query that can asked

```
PQ6: path(Src, DstA, P1), path(Src, DstB, P2), disjoint([P1, P2]).
```

Fig. 8. Are there two different paths with the same source and two different destination?

The closest Python implementation is much more complex.

```
for Src in interfaces:
    consume(Src)
    for DstA in interfaces:
        path(Src, DstA, P1)
        consume(P1)                 # the resource of P1 are used
        for DstB in interfaces:
            path(Src, DstB)
        release(P1)
```

Fig. 9. The Python code that is equivalent to the Prolog code in Fig.7.

The examples above show the fundamental difference between Prolog and Python. In Python each request from a user leads to a new algorithm that has to be specifically implemented while Prolog is more flexible on this aspect. The fundamental ambition of constraint Logic Programming is to separate modeling from algorithms as stated by Kowalski [14] with this pseudo equation:

$$Solution = Logic + Control$$

Where Logic is the declarative definition of the model and Control says how to find a solution. This is a strong advantage because most of the network constraints are represented in NDL naturally and thus map naturally to the logical structure of the language. In an imperative programming language the two aspects are fully interleaved. It is also important to understand that increasing the size of the queries (more free variables) leads to an explosion of the size of the search space. In order to cope with this search space explosion, a specific algorithm may be needed in both Python and Prolog. For example, the Prolog algorithm in Fig.6 does require additional variables for both tracking and backtracking the list of cross connects. In order to force a breadth first search algorithm instead of a depth first search algorithm would require additional tuning. Nevertheless the authors of [15] add the following pseudo-equation:

$$Control = Reasoning + Strategy$$

to emphasis the key difference between the reasoning employed to reduce the search space and the strategies or heuristics that helps in exploring the potentially interesting solution first. Within such conceptual framework it becomes much easier to understand where the complexity comes from as well as where are the degrees of freedom to optimize a specific constraint satisfaction problem as the multi-layer path finding.

VI. Future work

We plan to further investigate how the Prolog variants can help to solve the multi-layer path finding problem. There are two possible candidates: ECLiPSe [15] and FLORA-2 [16]. ECLiPSe integrates in the core of the language advanced constraint propagation techniques. Such techniques can prune the search space earlier than Prolog by detecting what exploration is useless as it will lead to an easy detectable conflict between constraints. ECLiPSe also supports, in a similar manner, the language construct for strategies and heuristics. Using ECLiPSe we desire to implement the multiple variants of the multi-layer path finding algorithm and compare their respective efficiency while dealing with representative networks. FLORA-2 proposes a support for object-oriented reasoning as well as the tabling extension, an interesting feature that prevents infinite recursion which may appear in Prolog.

A major problem in this area is that a segment of a shortest path may not be a shortest path in itself. This means that it is not possible to solve the multi-domain scenario locally, and it conflicts with the desire to minimize the amount of information exchange cross domains.

VII. Conclusion

In this paper we proposed a solution for the multi-layer path finding problem using the semantic network descriptions and a logical reasoning system (Prolog). The logical reasoning system provide a very concise method to implement complex path finding constraints. It also permits to fully take advantage of the semantic information that is encoded with the NDL model. While our work is still in an early state, we believe that declarative constraint logic programming is of great help not only for multi-layer path finding but also for problems like co-allocation of multiple resources in the context of Grids. The use of a declarative language serves as the glue to describe all these resources and their usage constraints in an unified framework.

References

[1] T. DeFanti, C. de Laat, J. Mambretti, K. Neggers, and B. St.Arnaud, "Translight: a global-scale lambdagrid for e-science," *Communications of the ACM*, vol. 46, no. 11, pp. 34–41, November 2003. [Online]. Available: http://doi.acm.org/10.1145/948383.948407

[2] L. R. Ford and D. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.

[3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[4] F. Dijkstra, J. van der Ham, P. Grosso, and C. de Laat, "A path finding implementation for multi-layer networks," *Future Generation Computer Systems*, vol. 25, no. 2, pp. 142–146, February 2009. [Online]. Available: http://staff.science.uva.nl/~fdijkstr/publications/ndl-pathfinding.pdf

[5] F. Kuipers and F. Dijkstra, "Path selection in multi-layer networks," *Computer Communications*, 2008. [Online]. Available: http://staff.science.uva.nl/~fdijkstr/publications/multilayer-pathselection.pdf

[6] J. J. van der Ham, F. Dijkstra, F. Travostino, H. M. Andree, and C. T. de Laat, "Using RDF to describe networks," *Future Generation Computer Systems*, vol. 22, no. 8, pp. 862–867, October 2006. [Online]. Available: http://dx.doi.org/10.1016/j.future.2006.03.022

[7] J. van der Ham, F. Dijkstra, P. Grosso, R. van der Pol, A. Toonk, and C. de Laat, "A distributed topology information system for optical networks based on the semantic web," *Journal of Optical Switching and Networking*, vol. 5, no. 2-3, pp. 85–93, June 2008. [Online]. Available: http://staff.science.uva.nl/~vdham/research/publications/0703-ApplicationsOfNDL.pdf

[8] "ITU-T Recommendation G.805: Generic functional architecture of transport networks," International Telecommunication Union, Tech. Rep., March 2000. [Online]. Available: http://www.itu.int/rec/T-REC-G.805/en

[9] "Unified functional architecture of transport networks," International Telecommunication Union (ITU), Recommendation ITU-T G.800, September 2007. [Online]. Available: http://www.itu.int/rec/T-REC-G.800/

[10] F. Dijkstra, B. Andree, K. Koymans, J. van der Ham, P. Grosso, and C. de Laat, "A multi-layer network model based on ITU-T G.805," *Computer Networks*, vol. 52, no. 10, pp. 1927–1937, July 2008.

[11] J. Wielemaker, "An overview of the SWI-Prolog programming environment," in *Proceedings of the 13th International Workshop on Logic Programming Environments*, F. Mesnard and A. Serebenik, Eds. Heverlee, Belgium: Katholieke Universiteit Leuven, december 2003, pp. 1–16, cW 371.

[12] J. Wielemaker, G. Schreiber, and B. Wielinga, "Prolog-based infrastructure for RDF: performance and scalability," in *The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida*, D. Fensel, K. Sycara, and J. Mylopoulos, Eds. Berlin, Germany: Springer Verlag, october 2003, pp. 644–658, lNCS 2870.

[13] B. T. Loo, T. Condie, M. Garofalakis, D. A. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: Language, execution and optimization," in *Proceeding of ACM-SIGMOD International Conference on Management of Data*, 2006.

[14] R. Kowalski, "Algorithm = logic + control," *Commun. ACM*, vol. 22, no. 7, pp. 424–436, 1979.

[15] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace, "Eclipse: A tutorial introduction," Imperial College London, Tech. Rep. IC-Parc-03-1, 2003-2008.

[16] G. Yang, M. Kifer, H. Wan, and C. Zhao, *Flora-2 : User's Manual*, 2008.